Subject: Re: C++ and CALL_EXTERNAL
Posted by jameskuyper on Thu, 29 May 2008 11:20:29 GMT
View Forum Message <> Reply to Message

mark.t.douglas@gmail.com wrote:
> After an entire evening wasted trying to get IDL to interface with a
> DLL I made, I thought I'd jot down the things that I wish I had known
> at the beginning, in the hope that it will be useful to someone,
> somewhere, sometime. I was using IDL 6.1 on Windows and Microsoft's
> Visual C++ Express 2008 compiler; the same procedure will work on
> other OSes, mutatis mutadis.
>
> OK, here we go. Suppose we have two functions, written in C++, that we
> wish to use from within IDL. We naively start with the following
> header:
>
```
> #ifndef NORMALS_H
> #define NORMALS_H
>
> namespace Normals
> {
>  __declspec(dllexport) double InverseCumulative(double x);
>  __declspec(dllexport) double Cumulative(double x);
> }
>
> #endif
```
>
> After building the DLL and moving it to IDL's working directory, we
> type the following into IDL:
>
```
> x = call_external("MyLib.dll","Cumulative",double(0.5),/all_value,/
> d_value,value=[0])
```
>
> It can't find the function! Why? Because the polymorphism and
> overloading features of C++ are usually implemented by mangling your
> nice function names into something that looks like a core dump.
> Examine your DLL with a program like PEDUMP to figure out what
> Normals::Cumulative() is now known as; I get ?
> Cumulative@Normals@@YANN@Z . That line noise encodes precise
> information about the argument types accepted by the function, believe
> it or not. Armed with this information, we type the following into
> IDL:
>
```
> x = call_external("MyLib.dll","?
> Cumulative@Normals@@YANN@Z",double(0.5),/all_value,/d_value,value=[0])
```

Wouldn't it be simpler to disable the name mangling by declaring the
functions as 'extern "C"' ? You can still use any feature of C++ that

you want, inside the definition of the function. Of course, you can't
use any C++ features in the function interface of an 'extern "C"'
function that are not also supported by C, but CALL_EXTERNAL probably
couldn't handle those features anyway.

Subject: Re: C++ and CALL_EXTERNAL
Posted by Robbie on Fri, 30 May 2008 00:46:33 GMT
View Forum Message <> Reply to Message

I would recommend making your own DLM's in preference to using
CALL_EXTERNAL and /AUTO_GLUE. Making your own DLM is a bit more
tedious but gives you more control in the way that IDL variables are
type cast.

If your keen to get dirty with C++ then I would recommend having a
look at my examples of converting Boost::MultiArray objects to and
from IDL Variables.

I take advantage of template specialization so that you only need to
ever use two functions:

const IDL_TYPE i = idl_cast_in<IDL_TYPE>(argv[0]);

and

idl_cast_out(argv[1],i);

Source code available from

http://barnett.id.au/idl/

Robbie

Subject: Re: C++ and CALL_EXTERNAL
Posted by mark.t.douglas on Fri, 30 May 2008 10:26:42 GMT
View Forum Message <> Reply to Message

On 29 May, 12:20, James Kuyper <jameskuy...@verizon.net> wrote:
> mark.t.doug...@gmail.com wrote:
>> After an entire evening wasted trying to get IDL to interface with a
>> DLL I made, I thought I'd jot down the things that I wish I had known
>> at the beginning, in the hope that it will be useful to someone,
>> somewhere, sometime. I was using IDL 6.1 on Windows and Microsoft's
>> Visual C++ Express 2008 compiler; the same procedure will work on
>> other OSes, mutatis mutadis.

```
>
>>  OK, here we go. Suppose we have two functions, written in C++, that we
>>  wish to use from within IDL. We naively start with the following
>>  header:
>
>>  #ifndef NORMALS_H
>>  #define NORMALS_H
>
>>  namespace Normals
>>  {
>>     __declspec(dllexport) double InverseCumulative(double x);
>>     __declspec(dllexport) double Cumulative(double x);
>>  }
>
>>  #endif
>
>>  After building the DLL and moving it to IDL's working directory, we
>>  type the following into IDL:
>
>>  x = call_external("MyLib.dll","Cumulative",double(0.5),/all_value,/
>>  d_value,value=[0])
>
>>  It can't find the function! Why? Because the polymorphism and
>>  overloading features of C++ are usually implemented by mangling your
>>  nice function names into something that looks like a core dump.
>>  Examine your DLL with a program like PEDUMP to figure out what
>>  Normals::Cumulative() is now known as; I get ?
>>  Cumulative@Normals@@YANN@Z . That line noise encodes precise
>>  information about the argument types accepted by the function, believe
>>  it or not. Armed with this information, we type the following into
>>  IDL:
>
>>  x = call_external("MyLib.dll","?
>>  Cumulative@Normals@@YANN@Z",double(0.5),/all_value,/d_value,value=[0])
>
>  Wouldn't it be simpler to disable the name mangling by declaring the
>  functions as 'extern "C"' ? You can still use any feature of C++ that
>  you want, inside the definition of the function. Of course, you can't
>  use any C++ features in the function interface of an 'extern "C"'
>  function that are not also supported by C, but CALL_EXTERNAL probably
>  couldn't handle those features anyway.
```

That would have worked fine and made life simpler for the two
functions I outlined here, certainly. However there are other things
in the DLL which are "proper" C++ so I elected not to use extern "C"
for the sake of consistency, as the DLL was designed as a C++ library
in the first instance. I probably should have mentioned this in the
original post!

Subject: Re: C++ and CALL_EXTERNAL
Posted by mark.t.douglas on Fri, 30 May 2008 10:45:05 GMT
View Forum Message <> Reply to Message

On 30 May, 01:46, Robbie <ret...@iinet.net.au> wrote:
> I would recommend making your own DLM's in preference to using
> CALL_EXTERNAL and /AUTO_GLUE. Making your own DLM is a bit more
> tedious but gives you more control in the way that IDL variables are
> type cast.
>
> If your keen to get dirty with C++ then I would recommend having a
> look at my examples of converting Boost::MultiArray objects to and
> from IDL Variables.
>
> I take advantage of template specialization so that you only need to
> ever use two functions:
>
> const IDL_TYPE i = idl_cast_in<IDL_TYPE>(argv[0]);
>
> and
>
> idl_cast_out(argv[1],i);
>
> Source code available from
>
> http://barnett.id.au/idl/
>
> Robbie

Making a DLM is overkill for this specific DLL - all I really want to
do is plot f(x) for f in the DLL, and I can wrap the call_externals
using stub functions to give a modicum of type safety:

```
function f,x
if ~ n_elements(x) eq 1 then return,0.d0
return,call_external('my.dll','function_name',double(x),/
d_value,values=[0])
end
```

That boost stuff, however, is very neat: thanks for the pointer :)

Subject: Re: C++ and CALL_EXTERNAL
Posted by jameskuyper on Fri, 30 May 2008 11:12:39 GMT
View Forum Message <> Reply to Message

mark.t.douglas@gmail.com wrote:
> On 29 May, 12:20, James Kuyper <jameskuy...@verizon.net> wrote:

...
>> Wouldn't it be simpler to disable the name mangling by declaring the
>> functions as 'extern "C"' ? You can still use any feature of C++ that
>> you want, inside the definition of the function. Of course, you can't
>> use any C++ features in the function interface of an 'extern "C"'
>> function that are not also supported by C, but CALL_EXTERNAL probably
>> couldn't handle those features anyway.
>
> That would have worked fine and made life simpler for the two
> functions I outlined here, certainly. However there are other things
> in the DLL which are "proper" C++ so I elected not to use extern "C"
> for the sake of consistency, as the DLL was designed as a C++ library
> in the first instance. I probably should have mentioned this in the
> original post!

I think that hard-coding the name-mangling scheme of one particular implementation of C++ in your IDL code is a bad idea. It makes your IDL code harder to read, and it might have to be changed if you use a different C++ compiler, or even a different version of the same C++ compiler. Declaring the function 'extern "C"' is a lot cleaner and more portable.

Don't let your concerns about "consistency" make your job harder than it needs to be. There's nothing wrong with using C++-specific features in the body of a C++ function with "C" language linkage. This is quite normal, because such functions usually serve as the interface between C++ code and non-C++ code. Nor is there any problem with having functions with "C" language linkage in the same translation unit as functions with "C++" language linkage.